Popular Computing

Volume 9 Number 2

February 1981

95

1	2	34	28560	67	67536
2	3	35	16170	68	138720
3	3	36	16839	69	4346
4	11	37	16872	70	260680
5	20	38	33980	71	349888
6	36	39	13104	72	47520
7	70	40	15960	73	91980
8	55	41	30668	74	186480
9	80	42	5880	75	7182000
10	295	43	13330	76	20520
11	121	44	230384	77	5544
12	119	45	62100	78	28067
13	130	46	2484	79	1161774
14	1176	47	1410	80	336000
15	300	48	264960	81	5320728
16	1440	49	16316	82	1883376
17	1428	50	12900	83	68475
18	576	51	27744	84	1585836
19	2128	52	2600	85	285090
20	320	53	2226	86	1475760
21	440	54	1156680	87	456228
22	483	55	75889	88	2265120
23	2622	56	11760	89	1226420
24	432	57	13680	90	75600
25	1900	58	493696	91	301938
26	780	59	14868	92	380880
27	4320	60	140400	93	539028
28	756	61	14640	94	78960
29	4466	62	135408	95	1938000
30	26400	63	232848	96	2428416
31	11284	64	25535	97	30264
32	2688	65	1034280	98	9603
33	20790	66	54648	99	67320
				100	273000
				101	8243620

The first 102 results of the

Penny Flipping II problem.

W

A Problem Solving Diary

This article will trace the steps involved in a straightforward computer solution to a non-trivial computing problem. The adventure being described is specifically in terms of an assembly-language program for the 6502 processor, but the language and machine are unimportant. What is important is the analysis and breakdown of the solution, and its organization.

Admittedly, not many people are interested in assembly language coding (which, on an Apple II, comes pretty close to coding in absolute hexadecimal). But then, on the face of it, you wouldn't think that there would be many people devoted to duplicate bridge (there are millions) or beer-can collecting (over a hundred thousand) or crossword puzzles.

Still, it is a fact that there are things to be done on the computer for which direct control of the machine is necessary. This is in contrast to working, say, in BASIC, where there are thousands of instructions between the user and the machine, and these instructions do things TO the user as well as FOR the user. The writer of those thousands of instructions made hundreds of decisions that the user must abide by, most of which he doesn't even know about.

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each.

Further, only through machine language can the inherent speed of the processor be capitalized on. The results given in Table W could be obtained in BASIC, perhaps, but only after many thousands of hours of execution.

The problem selected as suitable for this article first appeared in our issue number 23 as Penny Flipping II:

Given a stack of N pennies, initially sitting all heads up. Turn over (that is, flip) the top penny, then the bottom 2 pennies, then the top 3 pennies, then the bottom 4 pennies,...,and so on until it is the entire stack of N that is flipped. After every flip, test to determine if the stack has returned to all heads. Continue with the top 1, the bottom 2, top 3, and so on. Count the number of flips to return the stack to all heads.

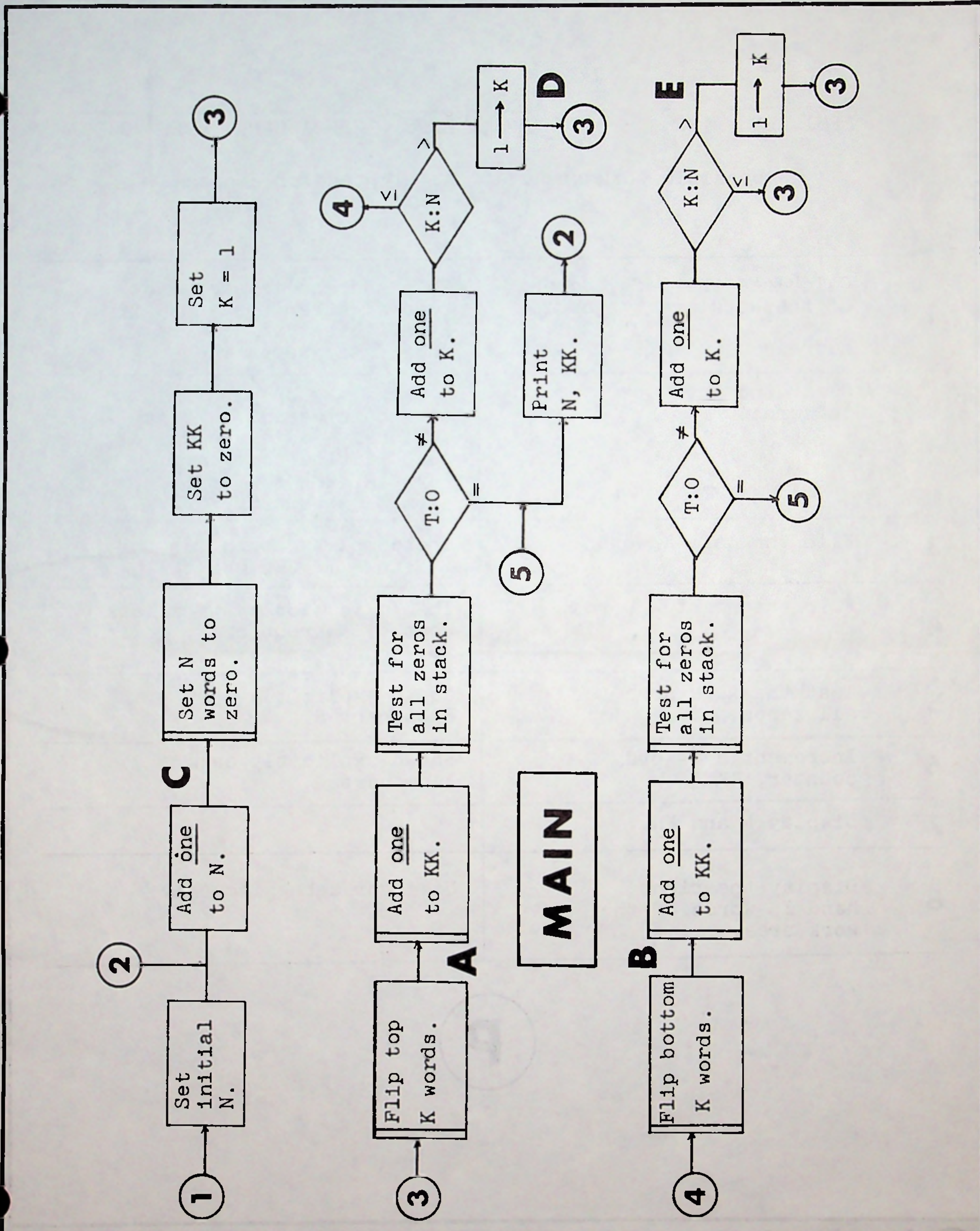
Not only does this problem lend itself nicely to what we wish to demonstrate, but an enormous amount of work has already been done on it and there is a conspicuous gap in the known results. Except for N = 58, the value of the function (that is, the number of flips to return to all heads) is known for all N from 1 to 64. The function is highly irregular and hence intriguing.

Consider: a stack of 98 pennies returns to all heads in 9603 flips, but a stack of 104 pennies takes over 18 million flips. This is irregular indeed.

0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0	The algorithm carried out for N = 5. A work area of 5 words is cleared to zero (representing all heads). The words at the left of the stack of five words represent the "top" of the stack; the words on the right represent the "bottom" of the stack. For N = 5, the process returns to all heads in 20 steps.

Digression: One of our goals over the years has been to find problem situations for which the computer is the essential tool for solution. Many such attempts have been thwarted by extremely clever analytic solutions. Indeed, it may be that someone could devise a formula for the solution of this problem, but its derivation would have to depend on a great deal of data like that of Table W. I submit that to acquire even a small portion of Table W, it is necessary to use a computer; no other tool will do.

The analysis of the proposed solution is summed up in the MAIN flowchart. The heart of the solution is the logic of flipping (essentially complementing the contents of a set of words and then inverting their order) K coins at either the "top" or the "bottom" of a set of N words, during the exploration of case N. The MAIN flowchart shows the overall (high level) logic of a solution (note that we are careful not to say "the" solution), which seems to lend itself to a group of sub-problems, listed in Figure P, each of which can be coded independently as subroutines. (Subroutine number one was an afterthought; it goes on the MAIN flowchart at C.)


Set initial N.
Add one to N.
C
Set N words to zero.
Set KK to zero.
Set K = 1
Flip top K words.
A
Add one to KK.
Test for all zeros in stack.
T:0
Add one to K.
K:N
1 → K
D
Print N, KK.
MAIN
Flip bottom K words.
B
Add one to KK.
Test for all zeros in stack.
T:0
Add one to K.
K:N
E
1 → K

BF81 BF82 BF83 BF84 BF85 ... BFFC BFFD BFFE BFFF | C000

A possible work area, at the high end of storage.

1	Calculate the left address of the work area; that is, address C000 - K.	This address is needed by several other routines; it is efficient to code it once as a subroutine.
2	Clear the work area to zeros.	For case N, only N words need be cleared. It is obviously easier to write a subroutine to clear, say, 127 words each time, from BF81 through BFFF.
3	Flip the left K words.	The complete logic for this is shown in Flowchart T.
4	Flip the right K words.	The logic closely parallels that of subroutine 3; it is not shown explicitly.
5	Test the work area for all zeros.	Shown explicitly as Flowchart R.
6	Increment a 4-word counter, KK	Shown explicitly as Flowchart Q.
7	Display N and KK.	
8	Display the right hand 15 words of the work area.	Used for debugging only.

P

In addition to segmenting the problem into a set of sub-problems (this is the most important use of subroutines) it is wise to prepare a map of the storage layout. Figure P shows the work area at its top. In addition, other words of storage can be allocated, something like this:

2000	2001	2002	2003	2004	2005
				K	CM

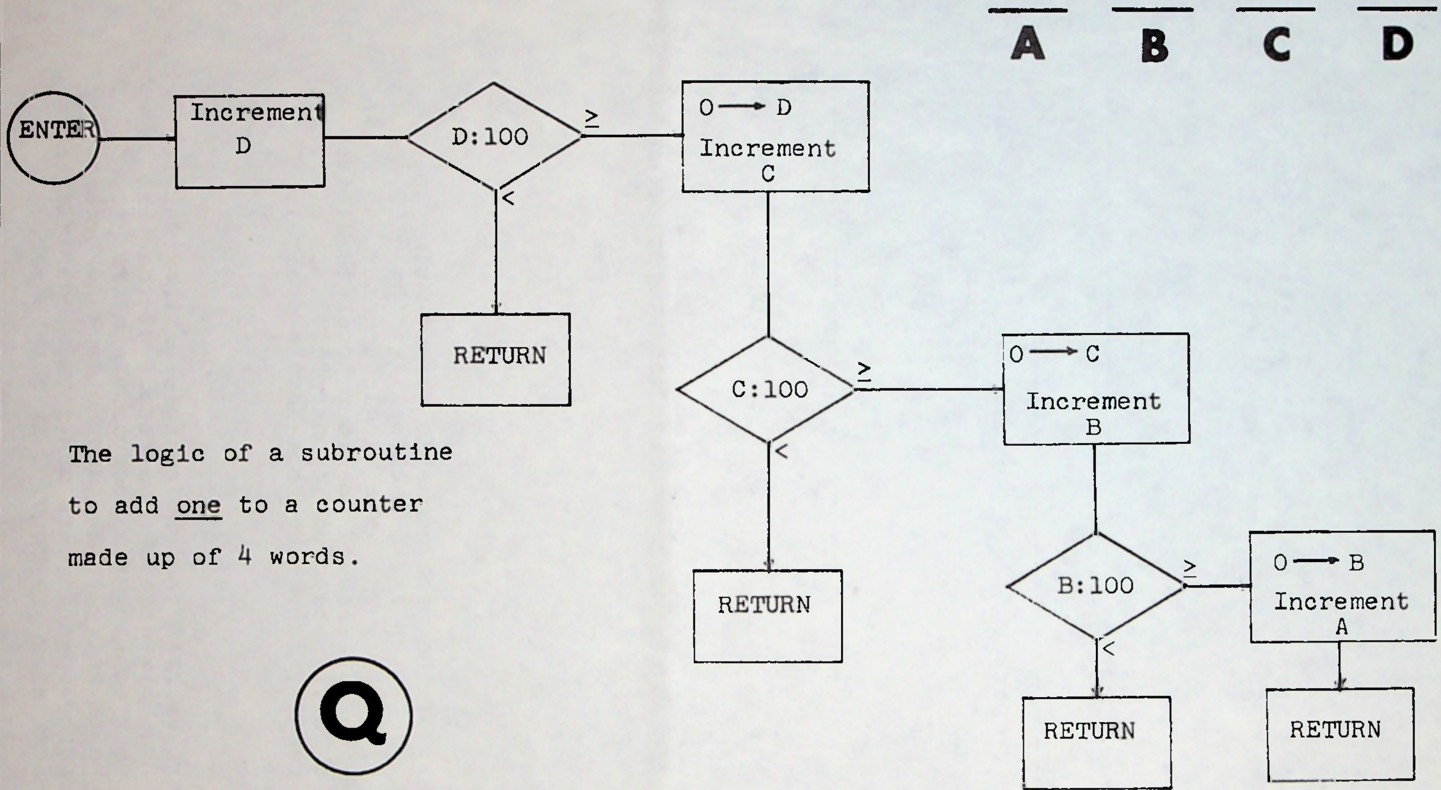
Four words needed for KK.

2006	2007	2008	2009	200A	200B
N	HI	LO	C2	C3	T

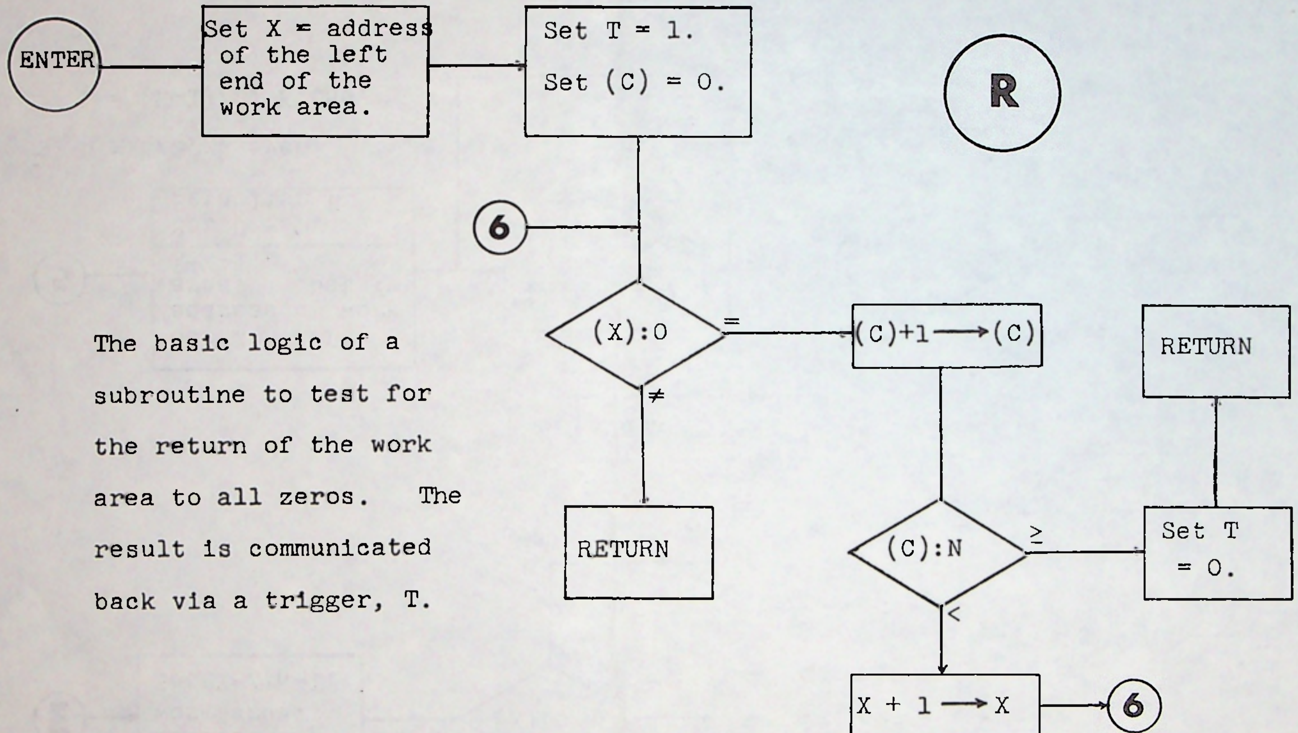
Portion of a storage map used in preparing an assembly language program for the Penny Flipping II problem.

The heart of the problem calls for counting to great heights. In an 8-bit machine like the 6502, it is expedient to use four words as a single counter, as shown in Flowchart Q. If each of the four words is limited to counting to 100 (instead of its natural limit of +127), the count is readily converted from hex to decimal.

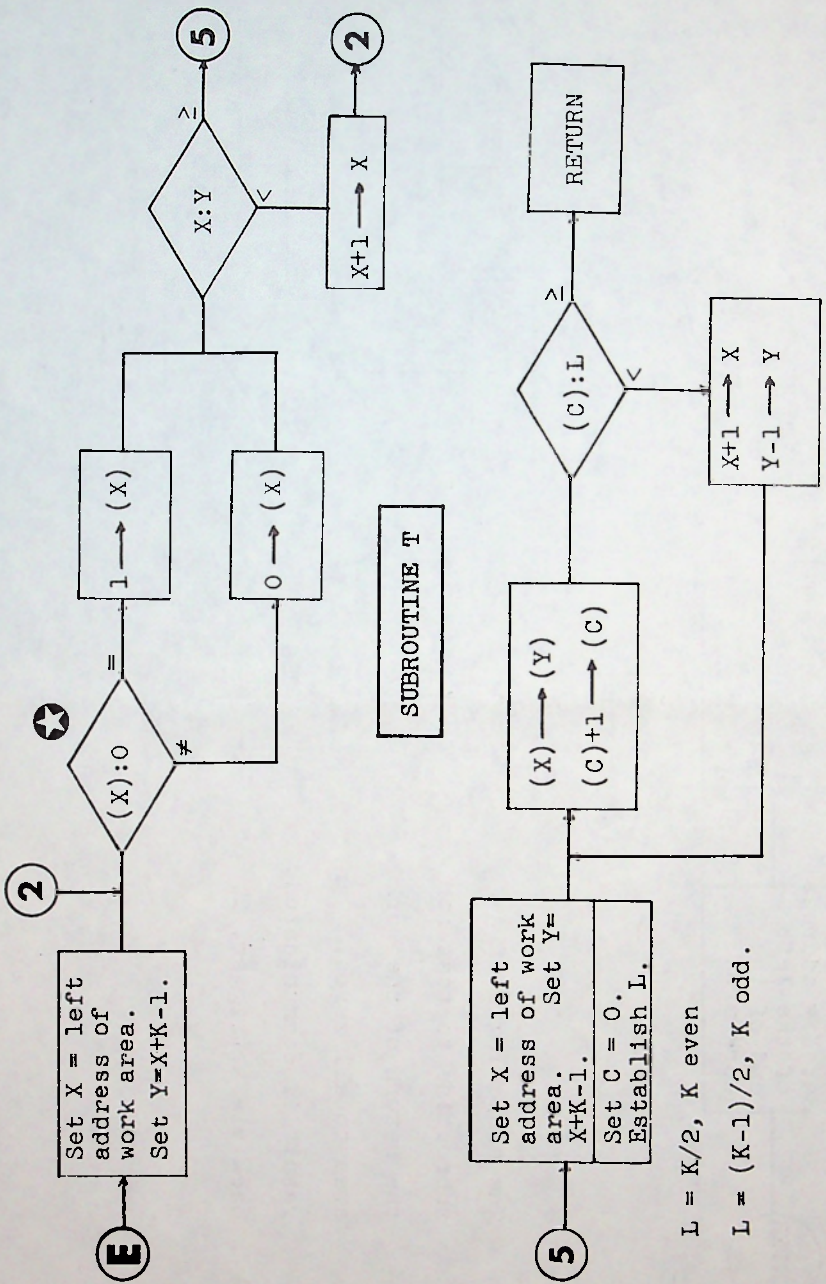
Flowchart R shows the basic logic of the test for all zeros in the work area. The result of the test is communicated back to the MAIN program via a trigger, T. T is set to one on entry to the subroutine and remains one if any word of the work area is non-zero. If all the words of the work area are zero, T is set to zero.



The logic of a subroutine
to add <u>one</u> to a counter
made up of 4 words.



The basic logic of a subroutine to test for the return of the work area to all zeros. The result is communicated back via a trigger, T.



E
Set X = left address of work area. Set Y=X+K-1.
2
(X):0
=
≠
1 ⟶ (X)
0 ⟶ (X)
X:Y
≥
<
5
X+1 ⟶ X
2
SUBROUTINE T
5
Set X = left address of work area. Set Y= X+K-1.
Set C = 0. Establish L.
(X) ⟶ (Y)
(C)+1 ⟶ (C)
(C):L
≥
<
RETURN
X+1 ⟶ X
Y-1 ⟶ Y
L = K/2, K even
L = (K-1)/2, K odd.

Subroutines 1 to 7 and the MAIN routine were written carefully and loaded into the machine. The signal to execute such a program (this one was about 250 instructions) is always exciting. Anything can happen, and that can include a perfect run right away--but long experience says that that eventuality is unlikely.

In this case, the program did not run properly. It ran, to be sure, but produced the result 4 for every value of N. The program was written to beep at the time of displaying each result, so the first trial run produced rapid beeps and endless 4's.

At this point, the breakdown of the solution into clearly defined subroutines really pays off. Even though the total program is clearly not working, the individual parts can be checked. Clearly, subroutine 5 (the test for all zeros in the work area) is already working fine, as is subroutine 7 (the display of results). It is easy to determine that subroutines 1, 2, 3, and 4 are already each doing their assigned task properly, but somehow the interactions are failing.

So subroutine 8 was added to the program, to display the high-order words of the work area. This subroutine was called at the places marked A and B on the MAIN flowchart. When the program was run again, the output from subroutine 8, for N = 10, showed:

$$\overline{1}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$$

$$1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ \overline{1\ 1}$$

$$\overline{0\ 0}\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1$$

$$0\ 0\ 0\ 0\ 0\ 0\ \overline{0\ 0\ 0\ 0}$$

and this pattern was repeated for each new value of N.

The bars over each line in the pattern indicate what should have been flipped. Actually, the flipping pattern seems to be top 1, bottom 2, top 1, bottom 2. The whole matter would be explained if the comparison shown at E in the MAIN flowchart was frozen on "greater than." Some study of the code revealed that that was exactly what was taking place; the comparison had been omitted entirely, but the branches based on the comparison were there.

Corrections and changes in machine language are usually easy to make by an old technique called "out to the woods and back." At the place in the program where a patch is to be inserted, a branch is overlaid, replacing an old instruction; this branch is to an unused portion of storage, preferably at the end of the routine involved. Then, out in this "woods" area, the stepped-on instruction is replaced, together with the necessary patching group of instructions, followed by a branch back to the instruction after the patch. It is all quite primitive, nostalgic, and thoroughly satisfying. It is good practice to label the "woods" end of the patch with a note as to where it came <u>from</u>.

All patches should be made on the coding sheets in a new color. The rule is: when any routine reaches four levels of Technicolor, it is time to scrap it and start over.

In our case, the missing COMPARE instruction was readily patched in, and another run was initiated, with N = 10. This produced immediate success:

0A 00 00 02 5F

The results were displayed in hexadecimal, of course, since that is the easy way to go, at least for a first try. The output translates quickly into:

10 295

which is correct (that is, it agrees with previously published results).

Subsequent answers were not so pleasing. An examination of the next 20 results (which took less than a minute to produce) showed that the answers for even values of N were all correct, but the answers for odd values of N were wildly wrong. Back to the MAIN flowchart: what's different about odd and even stacks of coins? It turns out to be this: an even stack usually ends at E, while an odd stack usually ends at D. The logic at D was wrong; it had been written to go to Reference 4, and it should go to Reference 3.

So yet another patch was made, and the program then flew correctly. The results shown in Table W beyond N = 64 (previously published) were simply a matter of CPU time. The time to obtain each result should be proportional to the product of N and the number of flips. An actual production run showed these numbers:

N	Number of flips, F	Time in Seconds	N times F divided by time
65	1034280	2550	26364
66	54648	138	26136
67	67536	172	26308
68	138720	357	26423
69	4346	12	24990
70	260680	688	26523
71	349888	936	26541

The logic of subroutine T shows one obvious short-cut at the place marked (*). The action of turning over a coin is being simulated by a single bit in a word of storage changing from zero to one or from one to zero. This is precisely the action of <u>complementing</u>, and is one of the chief uses of the exclusive OR operation, which is defined to be:

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 \end{array}$$

Thus, if the number 00000001 is OR'd with a word in storage, the low-order bit of the other word will change from zero to one or from one to zero.

The flowchart for subroutine T shows the complementing action being done the long way. This poses a problem; namely, how do the two approaches compare? The coding compares like this:

As pictured in flowchart T	Using the EOR command
Load word X	Load a one
Branch on equal (zero) to	EOR with word X
Load a zero	Store at X
Store at X	(Continue)
Branch to	
Load a one	
Store at X	
(Continue)	

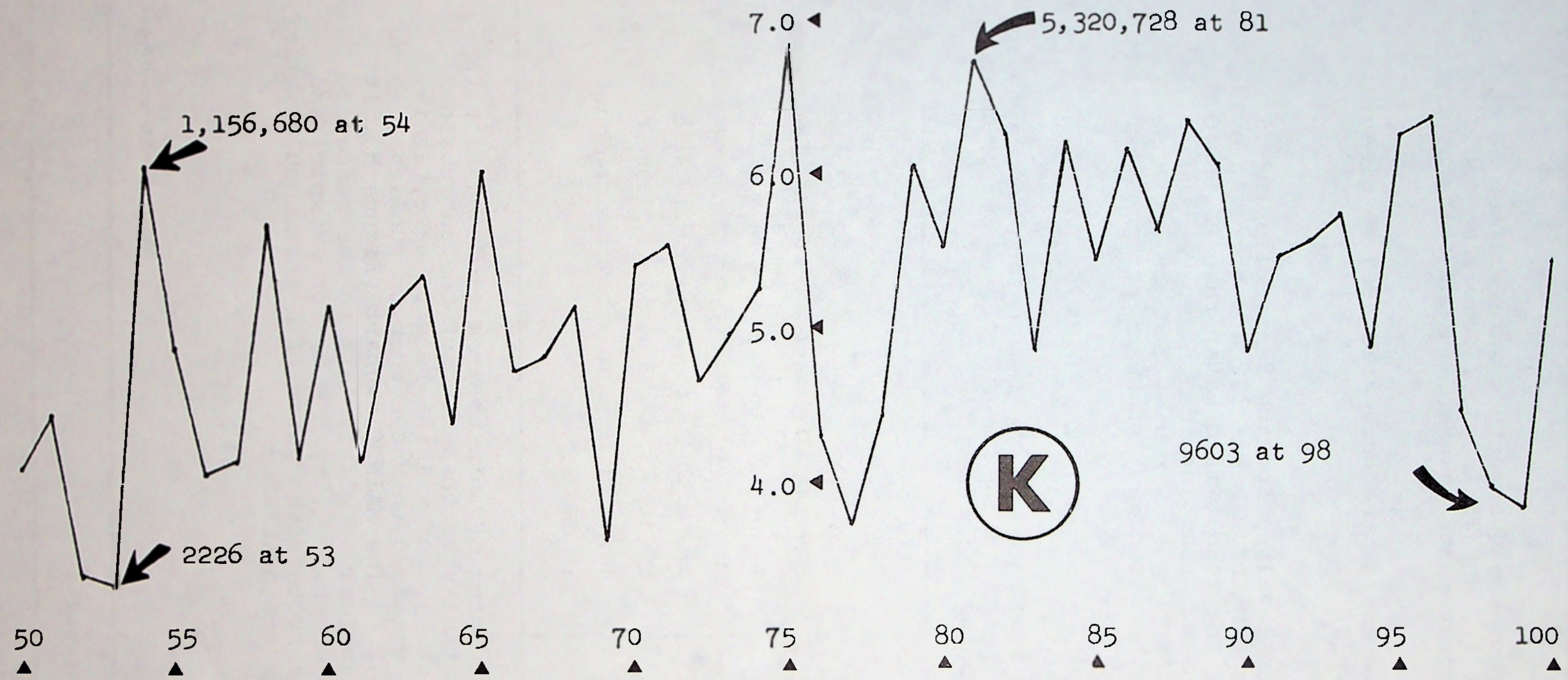
The coding on the left is simpler to follow; I would suggest such an approach during the debugging and testing phases of a new problem. Then, if it seems to pay, the shorter coding can be substituted; the program can then be re-tested, and the shortcut will then be effective.

In this case, the time difference between the two modes of attack amounted to 5.8%. If the simple-minded approach shown on the flowchart for subroutine T is the one first used, then the programmer must decide whether or not it is worth it to recode and retest the program for a 5.8% gain in speed.

It is axiomatic in the computing business that it is only after a program is tested and in production that its writer really understands how it should have been written. Thus, one's first attempt at a new program really should be just that: a trial program, made to be discarded. Looking at it slightly differently, a working program is an open invitation to write it again, only now we know how to do it right. The emergence of personal computers has made it possible for many people to enjoy this exquisite luxury.

When this program is rewritten, the following improvements should be considered:

1. Subroutines 3 and 4 (the action of performing flips of K coins on the top and bottom of a stack) should be each written as one continuous loop, instead of in pieces. As shown on Flowchart T, there are three distinct actions; namely, turning over the coins, copying the K words to another area of storage, and then copying them back in reverse order. This involved approach made the coding easier, but at the cost of inefficiency.



Penny Flipping II results for stacks of pennies from 50 to 100. The vertical scale is in common logarithms.

2. If some of the subroutines were written as <u>open</u> subroutines (that is, not linked to), some time could be saved. This improvement, however, is apt to be at the (1/10)% level.

3. Only N words of the work area need be cleared for each new value of N.

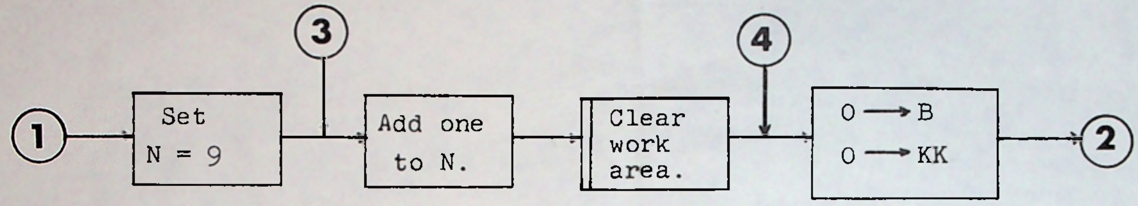
The functional values for the Penny Flipping II problem are now known through N = 112 and there is evidence that for N = 113, the result is greater than 28,000,000. The results are of no great intrinsic value (although the function, graphed in Figure K, is mysterious and hence intriguing), but the methods of producing correct programs are always worth exploring.

Rather than rewrite the program for the Penny Flipping II Problem, it was more attractive to write a new program for the Penny Flipping IV Problem:

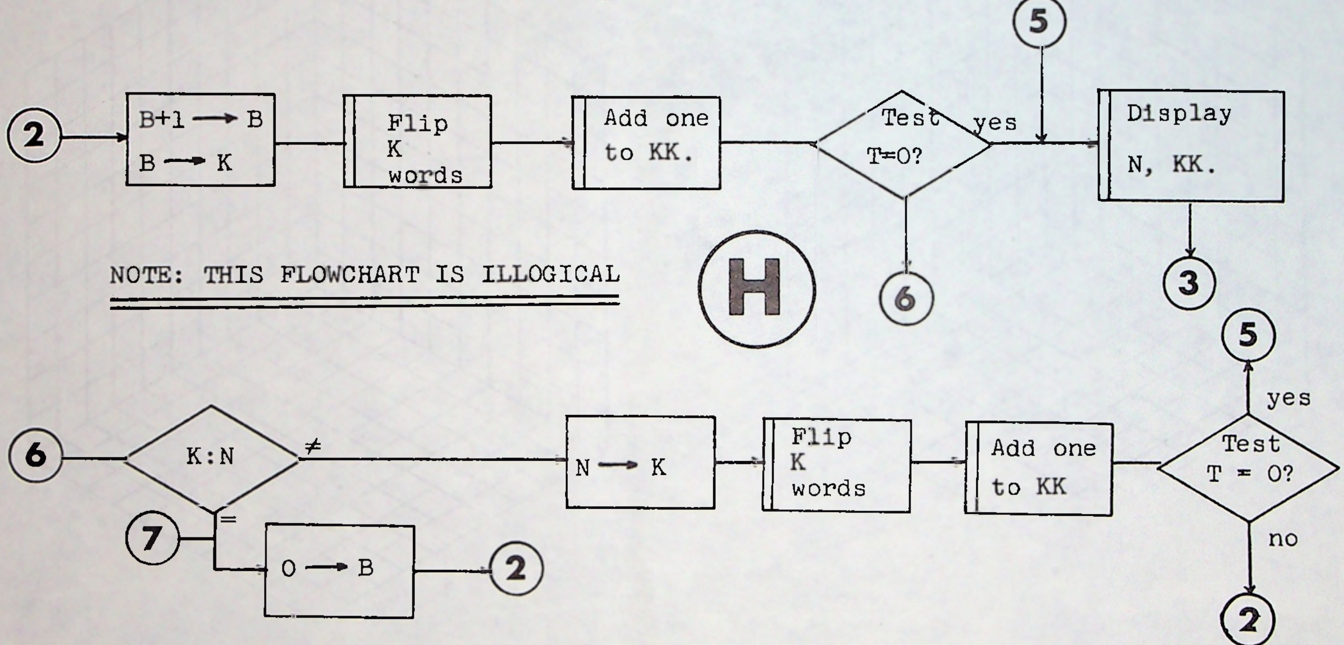
> *Given a stack of N pennies, initially all sitting heads up. Flip the top penny, then the entire stack, then the top 2, then the entire stack,...,until the top K pennies <u>are</u> the entire stack; then start over with the top 1, the entire stack, the top 2,...,and so on. Count the number of flips to return the stack to all heads.*

The first running of this program produced results (and this fact establishes that most of the subroutines are working properly), but all the results were, again, wildly wrong (as compared to known results published in our issues 25 and 29). The MAIN flowchart (Figure H), it turns out, is illogical. It is left to the reader to spot the logical error. (Try N = 8, for example, and trace through the successive values of K that should satisfy the conditions of the problem.)

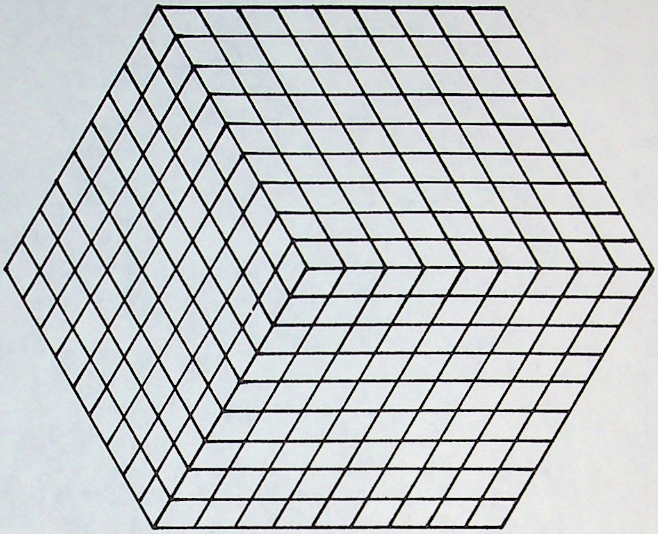
Fred Gruenberger



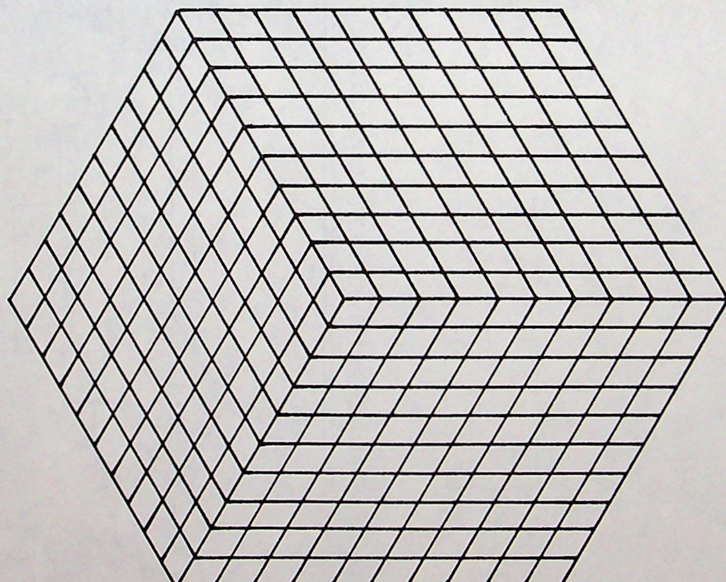
A first attack on the Penny Flipping IV Problem.

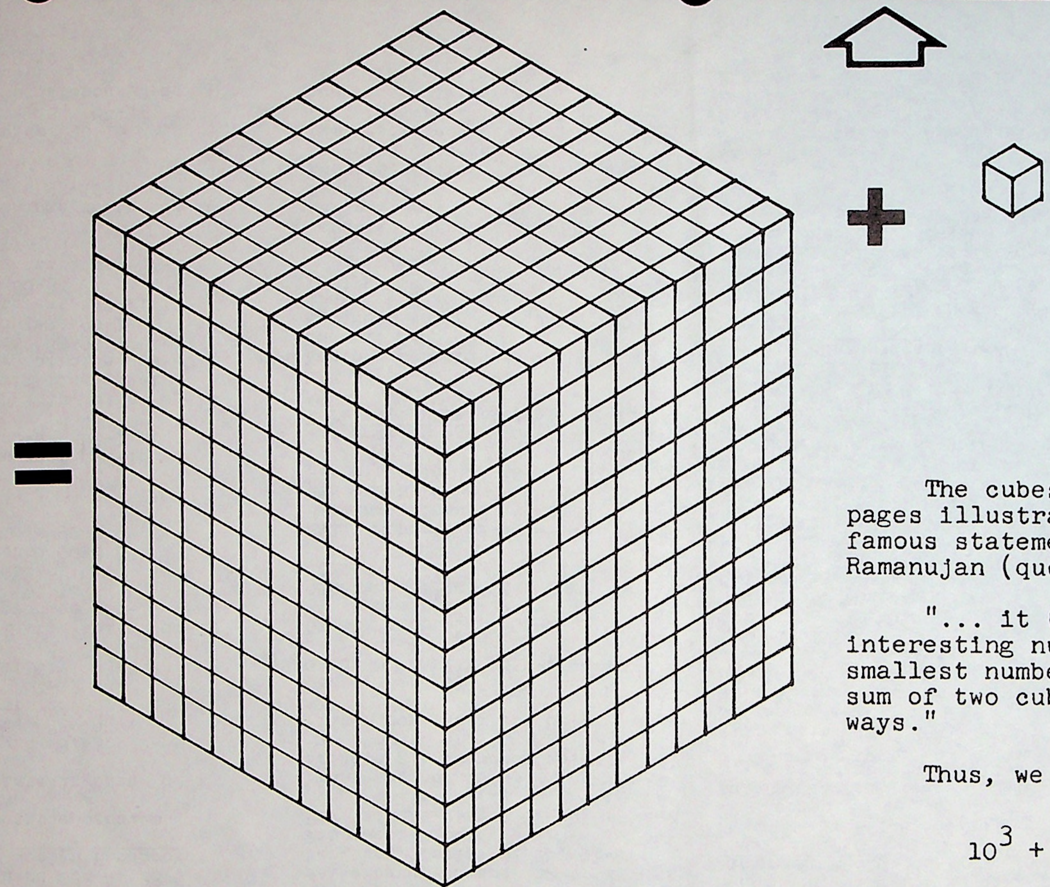


NOTE: THIS FLOWCHART IS ILLOGICAL



+





The cubes shown on these two pages illustrate graphically the famous statement of Srinivasa Ramanujan (quoted by G. H. Hardy):

"... it (1729) is a very interesting number; it is the smallest number expressible as a sum of two cubes in two different ways."

Thus, we have:

$$10^3 + 9^3 = 12^3 + 1^3$$

and we can demonstrate the fact with 1729 little cubes. No smaller number of cubes can be so arranged. It strikes me that this is as close as we can come to absolute truth; Ramanujan's statement requires no axioms or postulates and an absolute minimum of assumptions.

ANNUAL INDEX

Volume 8 1980
Issues 82 through 93
244 pages